Efficient multicast data transfer with congestion control using dynamic source channels

Vincent Lucas[1], Jean-Jacques Pansiot[1], Dominique Grad[1] and Benoît Hilt[2]

([1]) LSIIT - Université de Strasbourg - CNRS, France
{*lucas, pansiot, dominique.grad*}*@unistra.fr*
([2]) Laboratoire MIPS - Université de Haute Alsace
benoit.hilt@uha.fr

Research report - 23^{rd} april 2010

Efficient multicast data transfer with congestion control using dynamic source channels

Vincent Lucas[1], Jean-Jacques Pansiot[1], Dominique Grad[1] and Benoît Hilt[2]

([1]) LSIIT - Université de Strasbourg - CNRS, France
{*lucas, pansiot, dominique.grad*}*@unistra.fr*
([2]) Laboratoire MIPS - Université de Haute Alsace
benoit.hilt@uha.fr

April 28, 2010

Abstract

The most efficient receiver-driven multicast congestion control protocols use dynamic channels. This means that each group has a cyclic rate variation with a continuously decreasing phase. Despite promising results in terms of fairness, using efficiently these dynamic groups could be a challenging task for application programmers. This paper presents a sequencer which maps out application data to dynamic groups in an optimal way. Multiple applications such as file transfer or video streaming, can use this sequencer, thanks to a simple API usable with any buffer containing the most important data first. To evaluate this solution, we designed a file transfer software using a FEC encoding. Results show the sequencer optimal behavior and the file transfer efficiency, as a single download generates only little more overhead than TCP. Moreover, download time is almost independent of the number of receivers, and is already faster than TCP with 2 competing downloads.
Keywords: Multicast, Congestion control, FEC

1 Introduction

Most multicast flows such as IPTV are streamed at a constant rate using UDP. To fairly share the bandwidth with TCP streams, several multicast congestion controls have been proposed. Highly scalable ones are layered and receiver-driven, thus each receiver can adapt its reception rate to the network capacity. In the first propositions, such as *RLM* [1], *RLC* [2] or *FLID-SL* [3] , the layers correspond to multicast groups sent at a constant rate. Each receiver joins or leaves new multicast groups to respectively increase or decrease its reception rate. But, to avoid the multicast leave time latency problem, a new approach using dynamic rate groups has been proposed, such as *FLID-DL* [3], $WEBRC$ [4] or $M2C$ [5]. Each group begins to send at a high rate and slows down progressively until becoming quiescent. Thereby, a receiver has only to wait in order to reduce its reception rate. This solves the leave latency problem as a receiver only leaves a multicast group when it becomes quiescent. Despite promising results in terms of fairness, using efficiently these dynamic groups is a challenging task and as far as we know, no solution to this problem has been proposed yet. This paper presents a sequencer which maps out application data to dynamic groups in an optimal way. Multiple applications such as file transfer or video streaming, can use this sequencer, thanks to a simple API usable with any buffer containing the most important data first. Indeed, the application does not need to know anything of the dynamic layering mechanism and the sequencer is independent of the application as it works without knowing the data encoding scheme. To evaluate this solution, we designed a file transfer software using the sequencer and a FEC encoding. Results show the sequencer optimal behavior and the file transfer efficiency, as a single download generates barely more overhead than TCP. Moreover, download time is almost independent of the number of receivers, and is already faster than TCP with 2 competing downloads.

2 Addressed issues

An efficient massively scalable multicast congestion control uses several multicast groups with dynamic layering. Despite promising results in terms of fairness, using efficiently these dynamic groups is a challenging task and no solution to this problem has been proposed yet. The main purpose of this paper is to design a simple API, which enables to use dynamic multicast congestion control in an optimal way, and takes as parameter a buffer containing the most important data first. Therefore, a receiver which receives only $N\%$ of the source rate, obtains the first $N\%$ of the application buffer. Moreover, various applications can use this API without knowing anything about the dynamic layering used.

3 Multicast congestion control

This section gives the background on multicast congestion control, necessary to understand the design of an efficient sequencer. This description is based on the receiver-driven *Multicast Congestion Control* ($M2C$) [5] protocol, but the sequencer is also compatible with *FLID-DL* [3] and $WEBRC$ [4].

3.1 The source part

The source part is quite similar to $WEBRC$ and sends data using dynamic multicast groups (cf. figure 1). Each group begins to send at a high rate and then continuously decreases its rate. Once a group reaches a low rate threshold, it stops sending data and becomes quiescent. Therefore, a receiver just waits to decrease its reception rate and leaves only quiescent groups. Only the base group, which cumulative rate[1] is $g0$ in figure 1, never becomes quiescent. The source uses a time division named *Time Slot Index* (TSI), which changes each *Time Slot Duration* (TSD) seconds. For each change of TSI, K groups become quiescent and K new groups start at the K highest rates. Let us define sub_TSI the subdivision of the TSI corresponding to a duration of TSD/K. This notion of sub_TSI will be used by the sequencer.

3.2 The receiver part

This is the complex part of the multicast congestion control, see $M2C$ [5] for details. First, the receiver computes the fair rate corresponding to its experienced network conditions. Then, the receiver joins multicast groups to receive a rate close to the computed fair rate. Since $M2C$ uses hierarchical cumulative groups, a receiver can join group N only after joining all groups from 0 to $N-1$. Thereby, data sent to the lowest groups in the hierarchy are received by most receivers. Receivers of a static source design can only obtain a set of rates with a coarse granularity corresponding to the inter-groups rate granularity. Whereas, with dynamic source design a receiver can obtain almost any desired rate by joining groups more or less quickly, independently of the inter-group rate granularity. Indeed, figure 1 shows that if a receiver can get rate 1 by joining group at the shown time, another one can get rate 2 just by delaying its joins, and this with a smaller gap between rates 1 and 2 than between cumulative rates $g1$ and $g2$.

4 Principle of the sequencer

To illustrate the principle of the proposed sequencer, assume that we wish to send a video using a dynamic layering congestion control, such as each receiver obtains a video resolution corresponding to its reception rate. For each video frame, the video software generates a buffer containing the different resolutions. As explained before, the goal is to send the most important data to the lowest groups since most of receivers receives it. In these conditions, the source application has to map out its data in function of the continuously changing rate of each multicast group. This computation is too specific and too complex for the application level. The basic idea presented in this paper proposes to move this mapping complexity to a dedicated sequencer easily usable by various applications such as file transfer or video streaming. It provides a simple API, which only needs a hierarchically encoded buffer, without knowing the buffering scheme used by the application. The buffer has just to contain the most

[1] The cumulative rate for the group of level L corresponds to $\sum_{l=0}^{L}(rate_l)$.

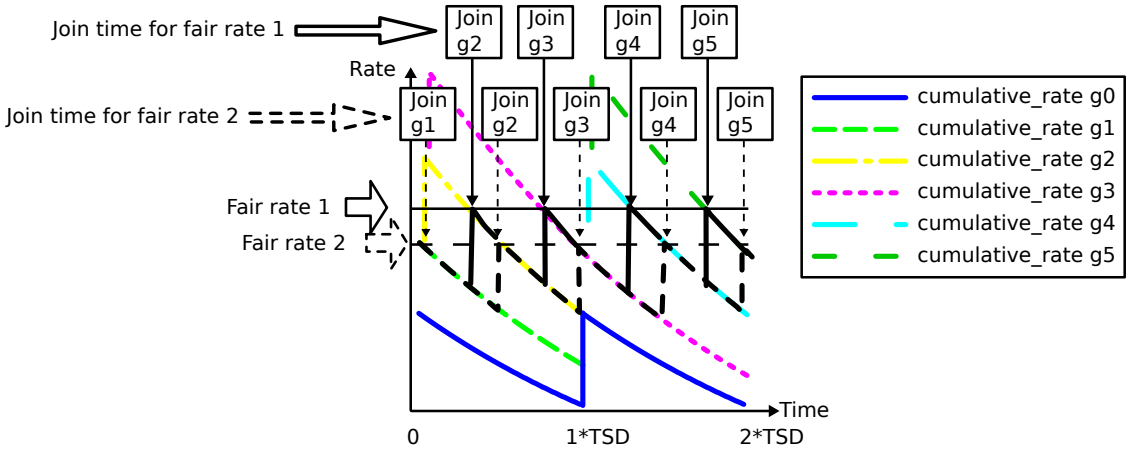


Figure 1: Fine rate granularity due to source dynamic design

important bytes at the beginning and the less important at the end. Then, the sequencer computes the amount of data that each multicast group can send, classifies and sequences the application buffer to send the most important data at the lowest groups. Thus, a receiver which receives $N\%$ of the source rate, obtains the first $N\%$ of the application buffer.

5 Proposition details

This section details how the sequencer splits incoming buffers into packets, orders and gives them to the multicast congestion control protocol, which schedules them.

5.1 Algorithm parameters

The sequencer provides a simple API easy to use, which only needs 3 parameters:

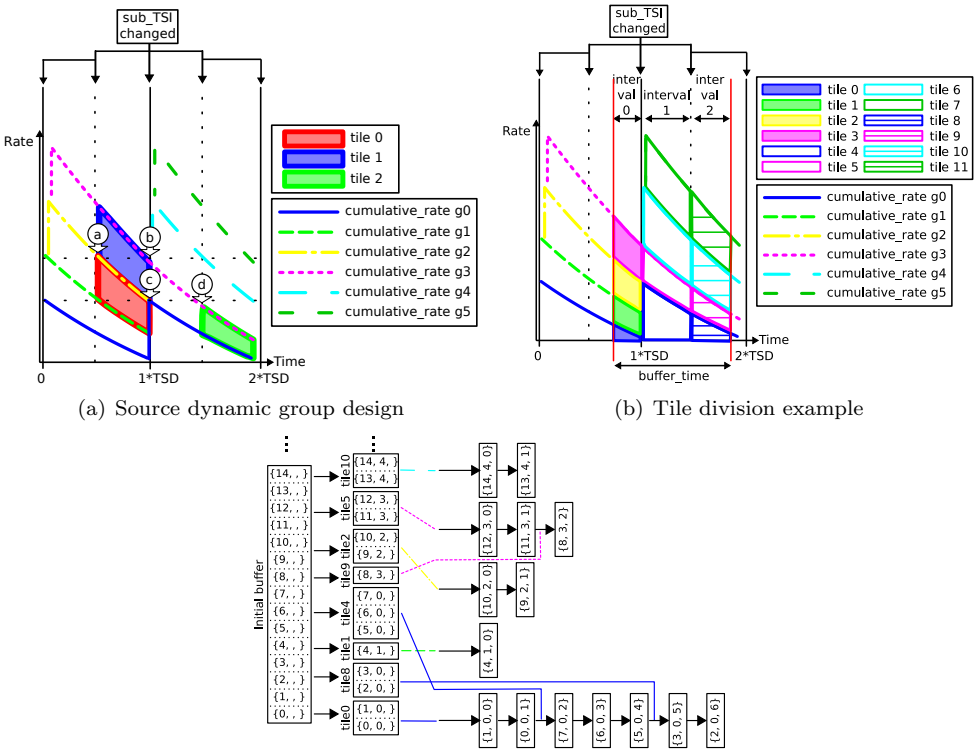
- The hierarchically encoded $buffer$ to send. It is important to note that the sequencer does not need to know the encoding scheme used.
- The time allowed to send the buffer ($buffer\_time$). For example, if the buffer is a video frame, $buffer\_time$ is the duration of the frame. Concerning file transfer or any other application where the $buffer\_time$ value is not obvious, it can be inferred by the application thanks to the hierarchy used. Since each receiver must at least receive the first level of the hierarchy, the $buffer\_time$ must correspond to the time needed to send the amount of data of the first level at the minimal congestion control rate[2]. Also, the $buffer\_time$ is independent of any parameter of the congestion control like TSD.
- The length of the buffer ($buffer\_length$). For a video codec, $buffer\_length$ depends on each image compression. For a file transfer, the ideal $buffer\_length$ is the amount of data sent in $buffer\_time$ at the maximal rate[2].

The sequencer must query some parameters from the congestion control too, such as the amount of data that can be sent for each group during a given time slot. Once all these parameters acquired, the sequencer computes the optimal distribution to send the most significant data to the lowest groups available.

5.2 Packet classifier

To send the most important data at the lowest group available, the sequencer creates a packet hierarchy dependent on time and on the group used. The packet hierarchy is subdivided in:

[2] The minimal and maximal rates of the congestion control, are fixed when the multicast congestion control session is initiated.



(a) Source dynamic group design

(b) Tile division example

(c) Algorithm example

Figure 2: Sequencer algorithm

- A coarse grain hierarchy, which splits each group into *tiles* and sorts all tiles in function of their minimal rate. A *tile* corresponds to a group during a *sub_TSI*. It takes TSD/K seconds for the minimal cumulative rate of group G_N to become lower than the maximal cumulative rate of group G_{N-1} during the same period. For example, figure 2(a) shows:
  - *tile*0 maximal cumulative rate ⓐ is equal to *tile*1 minimal cumulative rate ⓑ. This way, *tile*0 cumulative rate is lower or equal to *tile*1.
  - *tile*0 minimal cumulative rate ⓒ is equal to *tile*2 maximal cumulative rate ⓓ. This way, *tile*0 cumulative rate is greater or equal to *tile*2.

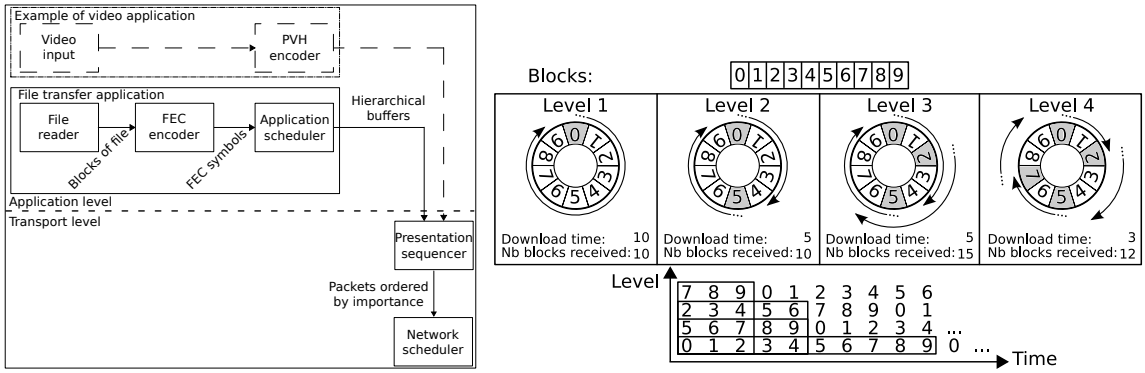
  The *tile* order of this example is: $tile2 < tile0 < tile1$. Meaning that the most important data must be send first in *tile*2, then *tile*0, then in *tile*1.
- A fine grain hierarchy to sort intra-*tile* packets. Depending on the join time, a receiver may get only the end of a *tile*, hence the hierarchy must be sorted in function of decreasing time: i.e. most important packet sent last.

5.3 Sequencer algorithm

The algorithm goal is to split the buffer into packets and to order these packets, sending the most important data at the lowest group available (i.e. figure 2(c)).
The first thing to do is to split the buffer into *Packet Data Units* (PDU). PDU will be identified by the triplet $\{j, g, s\}$ where:

- j is the PDU number[3]. With $j \in [0...J[$ and J the number of PDU contained in the application buffer. All PDU have the same size, except the last one if the $buffer\_len$ is not a multiple of the default PDU size.

[3] In practice, j is an offset relatively to the $buffer$. But for readability purposes, in this paper we use a PDU number corresponding to:
$PDU\_number := offset/PDU\_length$.

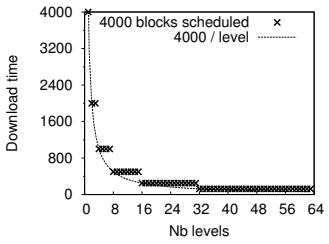


(a) Software data flow diagram

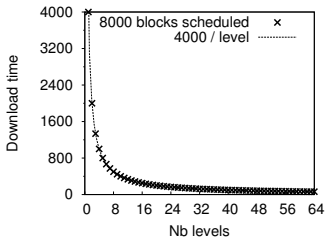
(b) Application scheduler algorithm

Level	nb blocks sent	nb blocks not sent yet
1	N	0
2	2*(N/2)	0
3	3*(N/4)	N/4
4	4*(N/4)	0
5	5*(N/8)	3*N/8
6	6*(N/8)	2*N/8
7	7*(N/8)	N/8
8	8*(N/8)	0
9	9*(N/16)	7*N/16
10	etc.	...

(c) Blocks sent before sending a duplicate



(d) 4000 blocks scheduled



(e) 4000 blocks scheduled with FEC ($k = 4000$, $n = 8000$, $\epsilon = 0$)

Figure 3: Application flow diagram and scheduler performances

- g is the group number to which PDU_j will be send. With $g \in [0...G[$ and G the number of groups.
- s is the packet sequence number relative to g. With $s \in [0...S_g[$ and S_g the number of packets that the group g can send in $buffer\_time$.

During $buffer\_time$ the same group can be partitioned into several tiles. Indeed, the algorithm splits the $buffer\_time$ into intervals defined by sub_TSI, as illustrated by figure 2(b). Then, it requests the corresponding number of packets $S_{g,i}$ that the group g can send during interval i. All tiles are sorted by their minimal cumulative rates, in order to send most important PDU to the tile having the minimal cumulative rate possible. For example in figure 2(c):

- Tile 0 will send PDU_j where $j \in [0...S_{0,0}[$.
- Tile 8 will send PDU_j where $j \in [S_{0,0}...S_{0,0} + S_{0,1}[$.
- etc.

For a given group, the tiles are sent following their chronological order: i.e. group 0 sends first tile 0, tile 4 and then tile 8. Finally, the packets of a *tile* are ordered following the fine grain hierarchy. Inside the same tile, a group continuously decreases its send rate, meaning that each tile must send less important PDU first, in order to send the most important data at the lowest rate available.

5.4 Sequencer header and receiver part

The sequencer header is composed of a buffer identifier and an offset relative to the application buffer. The receiver stores in a reception buffer all the datagrams until one arrives with a new buffer identifier. Then, the API provides to the application an access function to get a list of all buffer parts received or only the contiguous segment starting at offset 0.

6 File transfer software design

The sequencer can be used by various applications. To this end we implement an efficient file transfer with a fine grained rate tuning. Our goal is to show the performances and the ease of using a dynamic multicast congestion control thanks to the sequencer. This software uses a cyclic data transmission scheme named *carousel* loop [6] and sends data following the flow diagram from figure 3(a):

- The file reader splits a file into blocks and give them to the FEC encoder.
- The FEC encoder generates repair symbols from the file blocks also named source symbols. The source and repair symbols form the FEC symbols.
- The application scheduler orders the FEC symbols into hierarchical buffers and pass them to the API.
- The sequencer orders the hierarchical buffers into packets assigned to groups.
- The network scheduler is a part of the multicast congestion control. It schedules packets for each multicast group. $M2C$ [5] is used for this evaluation.

The sequencer coupled with the network scheduler can be reused for other applications and provides functionalities similar to TCP, except reliability which is managed by the FEC encoder and the *carousel*.

6.1 Application scheduler

The application scheduler, derived from [7], orders B blocks into buffers of N hierarchical levels (cf. figure 3(b)), such as the distance between two occurrences of a block is maximized and the number of redundant blocks received is reduced:

- The level 1 chooses block b for buffer b, with $b \in [0...B[$.
- The level n chooses block $b + x$ for buffer b. With $n \in [2...N[$ and x the block at the middle of the longest interval between two blocks of all lower levels. For example in figure 3(b), level 3 chooses block 2 since it is the block at the middle between block 0 and 5 distant of 5 apart.

Thus, download time is halved each time the receiver doubles the number of levels received (cf. figure 3(d)). Moreover, this property is valid whatever the first buffer b received. Fine rate granularity is provided by using a block size corresponding to the payload of a packet. Note that the levels used in the application scheduler are independent of network scheduler layers.

6.2 FEC encoder

A FEC code generates n symbols from k source symbols. The main advantage is that only $k + \epsilon$ (with ϵ small) distinct symbols suffice to recover the k source symbols. *Maximum Distance Separation* (MDS) codes ($\epsilon = 0$), such as Reed Solomon [8] codes, have a low encoding/decoding rate which limits the number of usable symbols. Whereas *Low-Density Parity-Check* ($LDPC$) codes ($\epsilon \neq 0$) release the limitation on the number of usable symbols and thus have an high encoding/decoding rate. Our file transfer software uses a $LDPCtriangle$ code [9] provided by the project "Planete-bcast" [10] library. Besides correcting packet losses, having numerous repair symbols can reduce the download time. The application scheduler analysis (cf. table 3(c)) shows that the number of unsent blocks before sending the first duplicated block is $< N/2$. Thus, having $n = 2 * k$ symbols improves the application scheduler and ables to reduce the downloading time for each new level received (cf. figure 3(e)).

7 File transfer evaluation

This section describes the testbed and the evaluation criteria used. Then, we analyze the application behavior for two different scenarii: 1 file transfer with 1 long competing stream and several file transfers with background traffic.

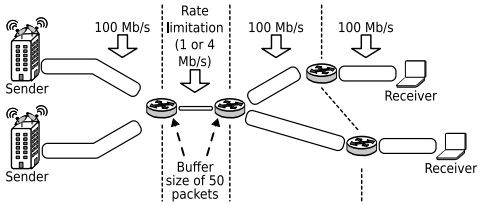


Figure 4: An example of the testbed with 2 senders and 2 receivers

7.1 Testbed

The file transfer software has been implemented [11] and evaluated on a testbed (cf. figure 4) composed of Linux routers running the *XORP* routing daemon (*PIM-SM*, *IGMPv3*) coupled with the *Token Bucket Filter* (*TBF*) module to set the bottleneck rate (1 or 4 Mb/s) and the buffer size (queues of 25 packets). As multicast congestion control has a slow rate adaptation, it is best suited to manage long lived streams. Then, the size of the files transferred must not be too small and we have chosen three files whose sizes are:

- 4 MBytes for a transfer lasting at least respectively 32 and 8 seconds at 1 and 4 Mb/s. This does not able *M2C* to converge to the fair rate and shows the behavior of the file transfer with important reception rate variations.
- 48 MBytes for a transfer lasting at least respectively 384 and 96 seconds at 1 and 4 Mb/s. This ables *M2C* to reach the fair rate.
- 99 MBytes for a transfer lasting at least respectively 792 and 198 seconds at 1 and 4 Mb/s. This file is the maximal sized manageable by our source and receiver *PC*s, due to memory limits.

The comparison is done between an unicast download client using *TCP new-RENO* congestion control algorithm with *SACK* option, and a multicast file transfer using our implementation [11].

7.2 Criteria

The file transfer performances evaluation criteria are based on:

- *time*: The number of seconds (s) used to download and recover the file.
- *gput*: The goodput corresponding to the application download rate (Kb/s).
- *tput*: The throughput corresponding to the link download rate (Kb/s).
- *loss*: The loss rate (%).
- *dup*: The duplicated symbols overhead (%) between the total number of FEC symbols received ($received\_symbols$) and the number of symbols needed by the FEC code to recover the file ($k+\epsilon$): $dup := (received\_symbols/(k+\epsilon)-1)*100$. The sequencer behaves well, if *dup* is close to 0.
- *sym*: The FEC symbols overhead (%) needed by the FEC code. It is the ratio between the number of source symbols (k) and the number of symbols needed by the FEC code to recover the file ($k+\epsilon$): $sym := ((k+\epsilon)/k-1)*100$.
- *head*: The header overhead (%) between the datagram or packet length ($packet\_length$) and the applicative data ($applicative\_data$) contained by the same packet:
  $head := (packet\_length/applicative\_data-1)*100$.
  Since $packet\_length$ is 1480 and that unicast and multicast carry respectively 1460 and 1448 bytes of $applicative\_data$, *head* is a constant overhead:
  - 1.4% for unicast streams, or 3.7% with ethernet and IP headers.
  - 2.2% for multicast streams, or 4.6% with ethernet and IP headers.

- *net*: The total network overhead (%). It is the ratio between the length of the file ($file\_length$) and the amount of data received at the link level ($link\_nb\_data$):
  $net := (link\_nb\_data/file\_length - 1) * 100$.
  This criterion represents a general estimation of the total overhead generated in the network and include overhead of *dup*, *sym* and *head*.

- *comp*: The computation time to recover the file after the last FEC symbol is received (%). It is the ratio between the *time* criterion and the time needed by the network to receive the file ($network\_time$): $comp := (time/network\_time - 1) * 100$. This criterion represents the time needed to restore the file once all the needed symbols are already received.

In the following results all these criteria are given with a confidence interval of 95% with a series of 20 tests for each experiment setup.

7.3 One file transfer with one long competing stream.

These tests show the file transfer (ft) behavior with a long competing stream:

- 1 *TCP file transfer* (TCP_ft) competing with 1 TCP stream.
- 1 *$M2C$ file transfer* ($M2C\_ft$) competing with 1 TCP stream.
- 1 *$M2C$ file transfer* ($M2C\_ft$) competing with 1 $M2C$ stream.

In all these tests, TCP *net* is less than 5%, mainly composed of the 3.7% of header overhead. Table 5(a) shows for 99 MBytes files that $M2C\_ft$ *net* is about 10%, mainly composed of at least 5.66% of *sym* and 4.6% of header overhead. Concerning the 4 MBytes file, $M2C\_ft$ *net* is between 13 and 16%. The increase of *net* is linked to a growth of additional FEC symbols required (8%). This can be explained as the ϵ extra symbols varies in function of the symbols received: all symbols have not the same importance. Moreover with small files, the *tput* variation is due to $M2C$ slow convergence to the fair rate, which confirms that $M2C\_ft$ is not suited for too short files. The small *dup* values prove the sequencer good behavior. TCP and $M2C$ streams use an *Additive Increase and Multiplicative Decrease* ($AIMD$) [12] mechanism producing cyclic losses. But, $M2C$ losses are more bursty, due to sudden rate variations when joining a new group. Regardless of the obtained throughput, the file transfer is not sensitive to the loss distribution and is quite as efficient with 3% of independent losses, as with 10% of bursty losses. Finally, download time differences are due to $M2C$ loose fairness when competing with another $M2C$ stream. The file transfer provides a fine grain rate as its behavior is not sensitive to the different bottleneck limitations and adapts itself to various *tput*.

7.4 File transfer with background traffic and multiple receivers.

These tests show the benefits of using $M2C\_ft$ when several receivers download the same file in a more realistic environment: 1 or 2 simultaneous downloads compete with background traffic composed of many short TCP connections such as those used for web browsing. These streams start times follow a Poisson process with an average of 10 streams per minute. Each stream sends an amount of data defined by an exponential distribution, such that 85% of the connections carry less than 6KB. Table 5(b) shows that for both TCP_ft and $M2C\_ft$, *net* is similar with 1 or 2 receivers. However, TCP_ft download time almost doubles when 2 receivers are involved, whereas $M2C\_ft$ download time remains stable. Thus, $M2C\_ft$ is able to take advantage of the multicast scalability, while unicast streams are forced to share the bandwidth with each other. Then, only 2 receivers are enough to make $M2C\_ft$ more advantageous than TCP_ft. To show that the benefit of using $M2C\_ft$ increases with the number of receivers of the same file, we have done another series of tests with 5 receivers downloading the same $48MBytes$ file competing with background traffic. Results shown in table 5(c) confirm $M2C\_ft$ performances. Indeed, these results point out that $M2C\_ft$ is almost independent of the number of receivers, while for TCP the download time increases linearly with the number of receivers.

8 Conclusion

This paper presents a new sequencer and API in order to ease the use of multicast congestion control with dynamic layering. Indeed, despite promising results in terms of fairness, using efficiently these

	4MBytes FILE			99MBytes FILE		
	TCP_ft vs. TCP	M2C_ft vs. TCP	M2C_ft vs. M2C	TCP_ft vs. TCP	M2C_ft vs. TCP	M2C_ft vs. M2C
1Mb/s bottleneck				1Mb/s bottleneck		
time(s)	79 (±10)	111 (±21)	125 (±62)	1737 (±32)	1760 (±112)	2330 (±366)
gput(Kb/s)	446 (±52)	318 (±54)	319 (±235)	477 (±9)	472 (±29)	358 (±50)
tput(Kb/s)	468 (±54)	365 (±65)	372 (±280)	495 (±9)	519 (±16)	391 (±25)
loss(%)	1.09 (±0.25)	2.84 (±0.61)	7.94 (±1.48)	0.81 (±0.04)	3.29 (±0.25)	6.6 (±0.45)
net(%)	4.0 (±0.41)	13.6 (±4.22)	15.05 (±6.76)	3.71 (±0.01)	9.56 (±3.94)	9.48 (±21.41)
dup(%)	N.A.	0.5 (±1.04)	0.77 (±0.92)	N.A.	0.08 (±0.05)	0.69 (±2.63)
sym(%)	N.A.	7.95 (±3.8)	9.1 (±6.14)	N.A.	7.14 (±4.12)	8.18 (±18.45)
comp(%)	N.A.	1.15 (±0.83)	1.16 (±0.93)	N.A.	0.45 (±0.14)	0.32 (±0.15)
4Mb/s bottleneck				4Mb/s bottleneck		
time(s)	20 (±2)	48 (±15)	62 (±48)	434 (±6)	438 (±29)	766 (±55)
gput(Kb/s)	1757 (±181)	767 (±290)	913 (±1139)	1910 (±26)	1896 (±122)	1090 (±76)
tput(Kb/s)	1909 (±194)	907 (±406)	1134 (±1679)	1986 (±26)	2134 (±136)	1223 (±93)
loss(%)	1.1 (±0.23)	2.46 (±0.81)	12.87 (±6.63)	0.67 (±0.09)	3.47 (±0.64)	8.97 (±0.98)
net(%)	4.99 (±1.46)	14.68 (±5.09)	15.64 (±5.3)	3.74 (±0.05)	10.58 (±0.55)	11.0 (±0.89)
dup(%)	N.A.	1.4 (±3.26)	1.8 (±3.06)	N.A.	0.1 (±0.04)	0.52 (±0.71)
sym(%)	N.A.	7.65 (±2.81)	7.95 (±2.81)	N.A.	5.66 (±0.53)	5.95 (±0.7)
comp(%)	N.A.	2.6 (±2.35)	3.79 (±7.1)	N.A.	1.79 (±0.32)	1.05 (±0.26)

(a) 1 file transfer versus 1 long stream

	4MBytes file		48MBytes file		99MBytes file	
	TCP_ft	M2C_ft	TCP_ft	M2C_ft	TCP_ft	M2C_ft
1 file transfer 1Mb/s bottleneck						
time(s)	38 (±1)	46 (±3)	419 (±2)	457 (±8)	876 (±3)	948 (±15)
gput(Kb/s)	913 (±19)	767 (±45)	941 (±4)	863 (±15)	947 (±3)	875 (±14)
tput(Kb/s)	972 (±3)	899 (±47)	978 (±3)	964 (±8)	983 (±3)	971 (±4)
loss(%)	0.76 (±0.09)	9.42 (±2.8)	0.29 (±0.03)	7.76 (±1.65)	2.56 (±2.32)	7.25 (±0.48)
net(%)	4.12 (±0.47)	13.81 (±1.92)	3.76 (±0.05)	10.69 (±1.65)	3.73 (±0.03)	10.15 (±1.28)
dup(%)	N.A.	0.24 (±0.25)	N.A.	0.22 (±0.22)	N.A.	0.2 (±0.04)
sym(%)	N.A.	7.82 (±1.65)	N.A.	5.84 (±1.6)	N.A.	5.78 (±1.12)
comp(%)	N.A.	3.07 (±1.75)	N.A.	0.91 (±0.21)	N.A.	0.77 (±0.16)
2 file transfer 1Mb/s bottleneck						
time(s)	73 (±5)	45 (±3)	827 (±16)	462 (±23)	1733 (±25)	976 (±66)
gput(Kb/s)	477 (±36)	771 (±59)	477 (±9)	853 (±40)	479 (±7)	851 (±55)
tput(Kb/s)	502 (±38)	899 (±52)	495 (±10)	960 (±45)	496 (±7)	964 (±48)
loss(%)	1.16 (±0.23)	8.19 (±2.82)	0.92 (±0.06)	7.75 (±1.28)	0.88 (±0.05)	7.18 (±1.13)
net(%)	4.04 (±0.85)	13.16 (±2.37)	3.73 (±0.08)	10.82 (±1.06)	3.71 (±0.04)	10.34 (±1.13)
dup(%)	N.A.	0.23 (±0.42)	N.A.	0.2 (±0.1)	N.A.	0.2 (±0.08)
sym(%)	N.A.	7.23 (±1.55)	N.A.	6.01 (±1.11)	N.A.	6.02 (±1.01)
comp(%)	N.A.	3.04 (±1.66)	N.A.	1.55 (±0.68)	N.A.	2.84 (±5.62)
1 file transfer 4Mb/s bottleneck						
time(s)	10 (±1)	17 (±4)	104 (±1)	124 (±5)	218 (±1)	253 (±8)
gput(Kb/s)	3457 (±285)	2118 (±457)	3793 (±35)	3195 (±139)	3808 (±13)	3282 (±106)
tput(Kb/s)	3948 (±20)	2688 (±744)	3967 (±6)	3708 (±154)	3967 (±5)	3743 (±131)
loss(%)	0.65 (±0.03)	3.24 (±2.72)	0.26 (±0.01)	8.38 (±6.08)	0.23 (±0.0)	7.7 (±2.59)
net(%)	7.85 (±0.55)	17.52 (±7.23)	3.96 (±0.17)	12.35 (±1.88)	3.79 (±0.02)	10.73 (±0.5)
dup(%)	N.A.	2.09 (±5.09)	N.A.	0.06 (±0.06)	N.A.	0.0 (±0.0)
sym(%)	N.A.	8.9 (±2.83)	N.A.	7.14 (±1.51)	N.A.	5.78 (±0.36)
comp(%)	N.A.	7.81 (±6.33)	N.A.	3.45 (±0.71)	N.A.	2.98 (±0.45)
2 file transfer 4Mb/s bottleneck						
time(s)	19 (±2)	17 (±9)	206 (±4)	127 (±7)	432 (±7)	262 (±11)
gput(Kb/s)	1879 (±173)	2083 (±741)	1913 (±39)	3108 (±165)	1921 (±30)	3172 (±135)
tput(Kb/s)	2029 (±179)	2737 (±1078)	1992 (±44)	3643 (±243)	1996 (±35)	3697 (±213)
loss(%)	1.08 (±0.31)	8.89 (±7.81)	0.85 (±0.07)	13.26 (±52.51)	0.82 (±0.04)	8.42 (±1.92)
net(%)	4.1 (±2.61)	17.76 (±7.3)	3.78 (±0.28)	11.33 (±0.93)	3.74 (±0.12)	10.81 (±0.48)
dup(%)	N.A.	1.64 (±3.88)	N.A.	0.0 (±0.0)	N.A.	0.0 (±0.0)
sym(%)	N.A.	9.61 (±3.1)	N.A.	6.21 (±0.82)	N.A.	5.84 (±0.42)
comp(%)	N.A.	11.4 (±8.51)	N.A.	5.28 (±2.45)	N.A.	5.18 (±2.28)

(b) 1 and 2 file transfers with background traffic

	TCP_ft	$M2C\_ft$	TCP_ft	$M2C\_ft$
	1Mb/s bottleneck		4Mb/s bottleneck	
$time(s)$	2055 (±30)	495 (±65)	512 (±8)	146 (±28)
$gput(Kb/s)$	192 (±3)	801 (±97)	770 (±13)	2733 (±467)
$tput(Kb/s)$	199 (±3)	921 (±129)	799 (±13)	3360 (±754)
$loss(\%)$	5.86 (±1.85)	9.86 (±9.5)	4.03 (±0.11)	8.17 (±3.36)
$net(\%)$	3.71 (±0.06)	11.64 (±7.27)	3.68 (±0.26)	11.38 (±1.41)
$dup(\%)$	N.A.	0.51 (±1.29)	N.A.	0.21 (±1.08)
$sym(\%)$	N.A.	6.49 (±1.79)	N.A.	6.06 (±0.93)
$comp(\%)$	N.A.	3.04 (±2.67)	N.A.	10.17 (±7.61)

(c) 5 file transfers with background traffic for the $48MBytes$ file

Figure 5: Experiments results

dynamic groups is a challenging task for application programmers and no solution has been proposed yet. The sequencer maps out application data to dynamic groups in an optimal way. Multiple applications such as file transfer or video streaming, can use this sequencer, thanks to a simple API which takes as parameter a hierarchically encoded buffer, without knowing the hierarchy application scheme. Thereby, receivers get all the most important data they can afford, since they join groups from bottom-up due to the cumulative hierarchy used by the multicast congestion control. To evaluate our proposition, we designed a file transfer using this sequencer, a multicast congestion control, an application scheduler and a FEC coder. The evaluation on a testbed shows the optimal behavior of the sequencer and the file transfer efficiency. Indeed, it only produces a slightly larger overhead than TCP for a single download. Furthermore, the multicast file transfer is highly scalable, almost independent of the number of receivers, and is more advantageous than TCP starting at only 2 receivers. Finally, this file transfer receives a fine rate granularity, which is totally independent of the group hierarchy used by the multicast congestion control. Moreover, each receiver can start downloading at any time independently of the source or other receivers. Currently, the transferred file size is limited by the PC memory. Future work will focus on the use of an interleaver to release this limitation. Lastly, the sequencer API is usable by various applications and is already integrated in a video streaming software using a PVH [13] codec with dynamic source channels.

References

[1] McCanne, S., Jacobson, V., Vetterli, M.: Receiver-driven layered multicast. In: SIGCOMM'96, Palo Alto, United States, ACM (1996) 117–130

[2] Vicisano, L., Crowcroft, J., Rizzo, L.: TCP-like congestion control for layered multicast data transfer. In: INFOCOM '98. Volume 3. (1998) 996–1003

[3] Byers, J., Horn, G., Luby, M., Mitzenmacher, M., Shaver, W.: FLID-DL: congestion control for layered multicast. Journal on Selected Areas in Communications (2002)

[4] Luby, M., Goyal, V.K., Skaria, S., Horn, G.B.: Wave and equation based rate control using multicast round trip time. SIGCOMM (2002) 191–204

[5] Lucas, V., Pansiot, J., Hilt, B., Grad, D.: Fair multicast congestion control (M2C). In: 12th IEEE Global Internet Symposium 2009, Rio De Janeiro, Brazil (2009)

[6] Peltotalo, J., Peltotalo, S., Harju, J., Walsh, R.: Performance analysis of a file delivery system based on the FLUTE protocol. International Journal of Communication Systems (2007) 633–659

[7] Birk, Y., Crupnicoff, D.: A multicast transmission schedule for scalable Multi-Rate distribution of bulk data using Non-Scalable Erasure-Correcting codes. In: INFOCOM'03, San Francisco (March 2003)

[8] Rizzo, L.: Effective erasure codes for reliable computer communication protocols. SIGCOMM Comput. Commun. Rev. **27**(2) (1997) 24–36

[9] Roca, V., Khallouf, Z., Laboure, J.: Design and evaluation of a low density generator matrix (LDGM) large block FEC codec. In: Group Communications and Charges. Number 2816 in LNCS. (2003) 193–204

[10] planete-bcast: Tools for Large Scale Content Distribution: http://planete-bcast.inrialpes.fr/ (2006)

[11] mutual: multicast file transfer application: http://sourceforge.net/projects/mutual/

[12] Yang, Y., Lam, S.: General AIMD congestion control. In: International Conference on Network Protocols. (2000) 187–198

[13] McCanne, S., Vetterli, M., Jacobson, V.: Low-complexity video coding for receiver-driven layered multicast. Journal on Selected Areas in Communications (1997)